Department of Electrical and Computer Engineering

332:423 Computer And Communication Networks Fall 2011

Assignment 3: TCP Tahoe with More Realistic Time Simulation and Packet Reordering

Group 3:

Craig Gutterman

Bartosz Agas

Ruslan Fridman

December 6th , 2011

Project Contribution:

Ruslan Fridman (33⅓ %)

- Analysis of Results
- Coding
- Debugging
- Algorithm
- Report

Bartosz Agas (33⅓ %)

- Analysis of Results
- Coding
- Debugging
- Algorithm
- Report

Craig Gutterman (33⅓ %)

- Analysis of Results
- Coding
- Debugging
- Algorithm
- Report

Table of Contents

Introduction

Transmission Control Protocol (TCP) was designed to help provide a reliable connection over the internet between hosts by assuring that data would be transmitted in an ordered fashion. As more and more users communicate with each other online, congestion becomes a serious issue. TCP Tahoe was one of the original congestion avoidance algorithms used to adjust the size of the data being sent out to help provide stable and reliable transmission.

In order to completely understand how TCP Tahoe works and the results seen for the various simulations in this project, the following parameters must be understood:

ACK (Acknowledgment) – signal passed from a receiving host to a sending host to acknowledge a received packet

dupACK (Duplicate Acknowledgment) – when a received ACK is a duplicate of a previous ACK. This informs the sender of dropped/lost TCP segments

MSS (Maximum Segment Size) –restricts the size of each segment that can be sent by the original host

CongWindow (Congestion Window) – sender's estimate of the buffer space available in the router for transmission

FlightSize (Flight Size) – amount of data sent, but not acknowledged

- FlightSize = LastByteSent – LastByteAcked

EffectiveWindow (Effective Window) – max amount of data that is currently allowed to be sent

- EffectiveWindow = min {CongWindow, RcvWindow} – FlightSize

SSThresh (Slow Start Threshold) – boundary value used to determine whether to use the slow-start or congestion avoidance mechanism

RTT (Round-Trip Time) – time it takes for a packet to be sent out plus the time for the ACK signal to be received by the sender

TCP Tahoe functions by implementing a congestion window in order to limit the amount of lost packets. It begins by initializing a 'slow start' function which begins by setting a small congestion window that increases by one MSS with every successful transmission of data. With each increase of the MSS, the congestion window doubles for every RTT. Segments sent out of order are accounted for by the sender through the transmission of dupACKs by the receiver. This continues till either one of two things happen, the congestion window reaches the SSThresh value or the sender receives three dupACKS. If the SSThresh is reached, TCP Tahoe enters a congestion avoidance state which has both the sender and receiver communicate in order to respond to heavy congestion over the network. If the three dupACKs are received, this means that there were packets lost in one of the previous transmissions. When packets are lost, Tahoe performs a fast retransmit, which sets a timer to help account for lost packets. In addition, it reduces the MSS size by 1 and re-initializes the slow start method mentioned earlier.

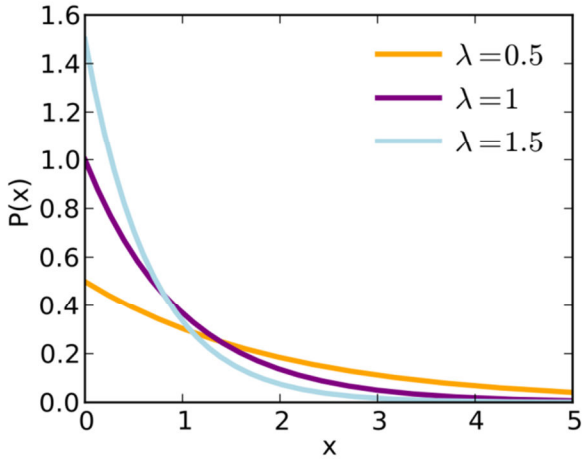
The objective of this project is to design a more realistic time simulation for TCP Tahoe by taking any packets ready to be sent out and re-ordering them based on a randomly assigned delay. This is done through the use of an exponential distribution. It is used in order to assign newly received packets with a delay based off a user input average. Since packets arrive in a random recurring independent event sequence, this distribution was the best fit for the project.

An exponential distribution ensures that most packets will be sent with little to no delay, but it allows for longer delays to be possible as well, which is how real systems behave. A brief overlook of the exponential function is seen below with its probability density function and plot.

$$f_X(x) = \begin{cases} \lambda e^{-\lambda x} & if\ x \in \mathbb{R}_X \\ 0 & if\ x \notin \mathbb{R}_X \end{cases}$$

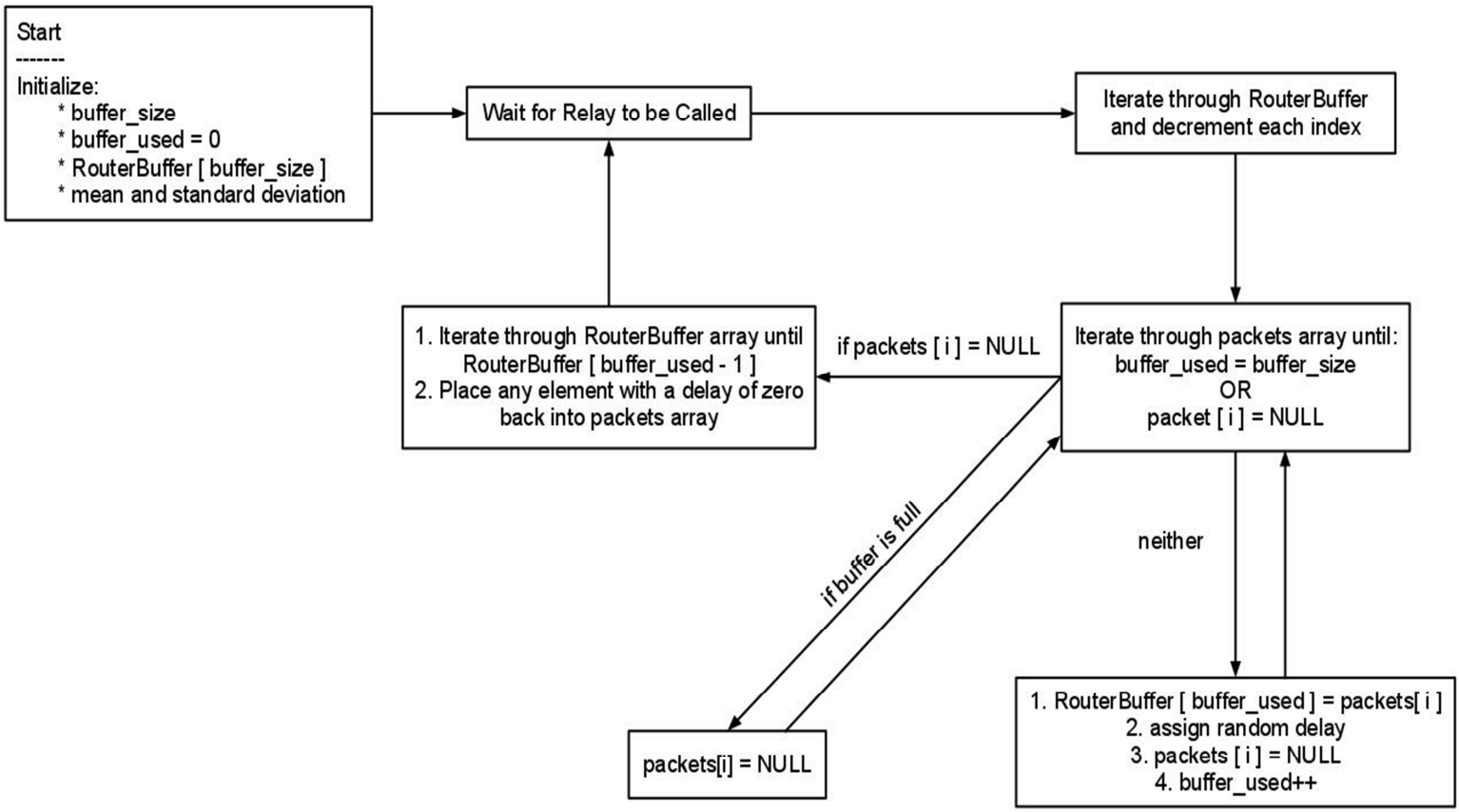
$$where\ \lambda > 0$$

$$\lambda\ is\ a\ rate\ parameter$$



Software Design

Algorithm Used:



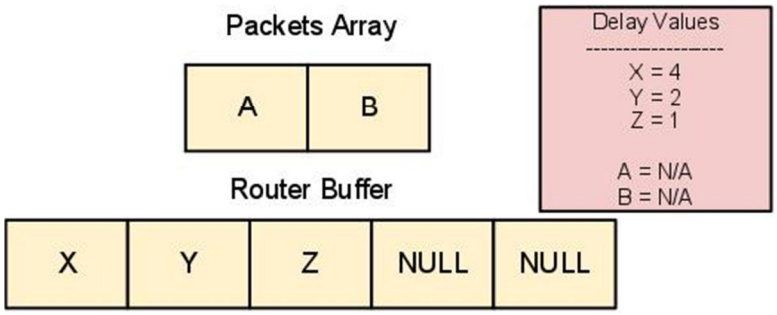
The algorithm begins by initializing the RouterBuffer array, setting buffer_used to zero, and having the user define buffer_size, mean, and standard deviation. Once that is done, it will wait for packets to be sent to the router. When received, it will decrement any packets currently in RouterBuffer to give them higher priority over new packets.

With the newly received packets, the algorithm continues by looping through the array of packets until the buffer fills up (buffer_used = buffer_size) or it hits the end of the packets array (packets[i] = NULL). With every iteration, each index of the packets array will be added to the end of the buffer and assigned a random delay value based on the user input mean and deviation. If the buffer fills up, it will just drop the remaining packets until it hits the end of the packets array.

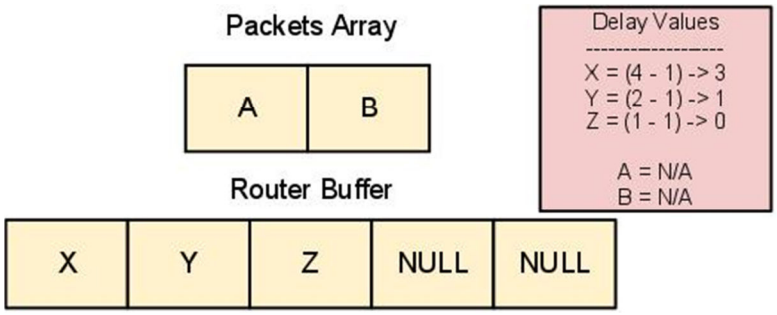
Once it hits the end of the packets array, it will iterate through RouterBuffer and then place any element with a delay of zero back into the packets array to be sent out. Once that is complete, the algorithm returns to the previous stage of waiting.

A quick run through of the implementation is as follows:

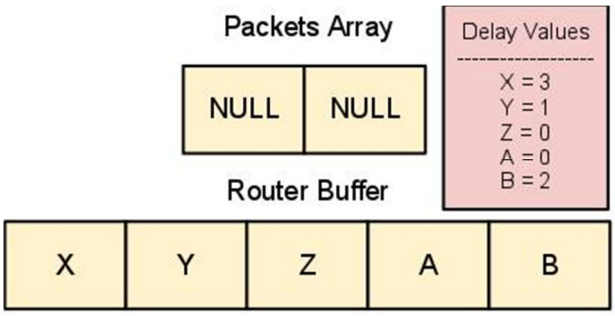
1.) Packets A and B arrive to see that X, Y, and Z are currently in the Router Buffer.



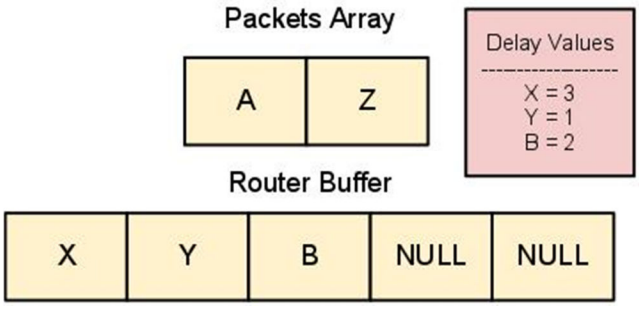
2.) Decrement the delays for the elements currently in the buffer



3.) Insert A and B into the buffer and assign then a random value



4.) Remove the elements with a delay of zero and insert them into the packets array for transmission



Implementation

This project was implemented in Java, using existing code as a starting point. The available code describes a router, a sender and a receiver communicating using TCP Tahoe constrained by the router buffer size. The code consists of many classes but we were primarily interested in the Router class. This class describes the behavior of a router facilitating communication between a sender, represented by the Sender class, and a receiver, represented by the Receiver class. The code assumes instantaneous and perfect transmission of acknowledgments without going through a router, therefore Router only handles data packets.

The Router class was almost completely rewritten but the methods were left intact. Router has two methods, getBottleneckCapacity() and relay(), as well as a constructor. The first method, getBottleneckCapacity(), was left unaltered but relay() and the constructor were heavily modified. The constructor initializes all of the variables with values passed to it or with appropriate starting values. The method relay() is called when packets need to be transmitted; It takes as input an array of TCPSegment's, which represent the incoming packets. For each TCPSegment it will assign the segment a random delay (exponentially distributed) and place it into RouterBuffer, a Vector. RouterBuffer was made a Vector rather than an array to simplify adding and removing packets. Arrays have a fixed length and they must be indexed by location meaning we would need an explicit pointer to the last element in the array; Vectors on the other hand are automatically resized if needed and data can be added to the end without explicitly keeping track of the last element location. These features are convenient but our main reason for using the Vector data structure instead of an array is the ability to remove data elements without having to explicitly shift the remaining values to fill in the gap. In an array, removing the nth element would require you to shift all of the elements in the n+i position, where i=1,2,3,4…, down by 1. Vectors on the other hand will do this for you when you use the remove() function.

The exponential distribution is derived from a uniform distribution generated with the Java math library. The delay is calculated with the following formula:

$$Delay = [-mean * \ln(r)]$$

Here mean is the user specified mean for the distribution, r is a random number uniformly distributed between 0 and 1 and the brackets represent round. In this formula the logarithm converts the uniform distribution r into an exponential distribution while the mean acts as a scaling factor. This delay function is a discrete value; therefore this is not a true exponential distribution because exponential distributions are continuous. Our delay needs to be an integer value thus we round the original exponential distribution.

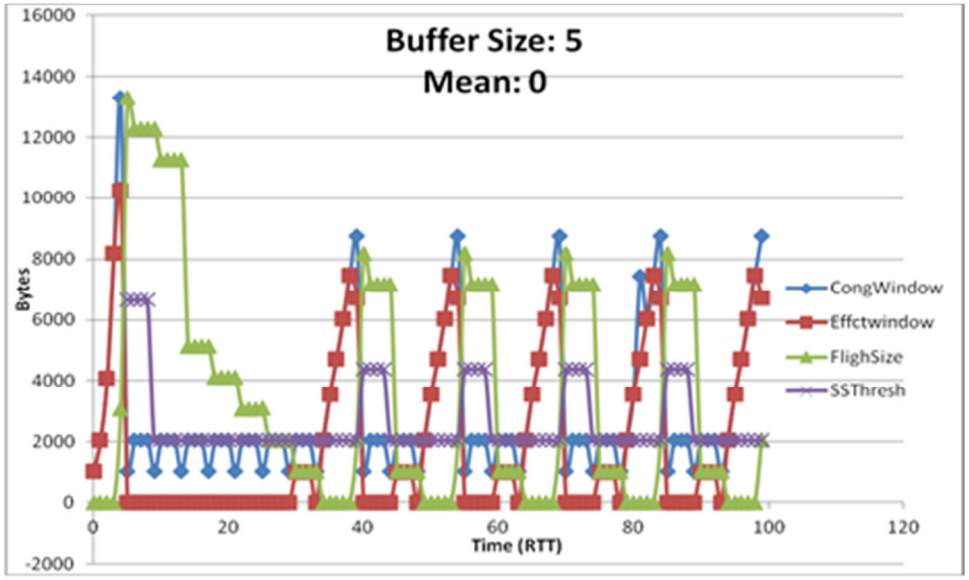
In addition to storing packets and assigning delays the Router class records pertinent data in text files to simplify analysis. It generates two text files, delay and droppedpackets. The delay file has the Iteration, total packets buffered, and the specific packet name with its corresponding delay. The droppedpackets file has the iteration number and how many packets were dropped in that iteration. These files were used to construct the various plots needed to analyze the data.

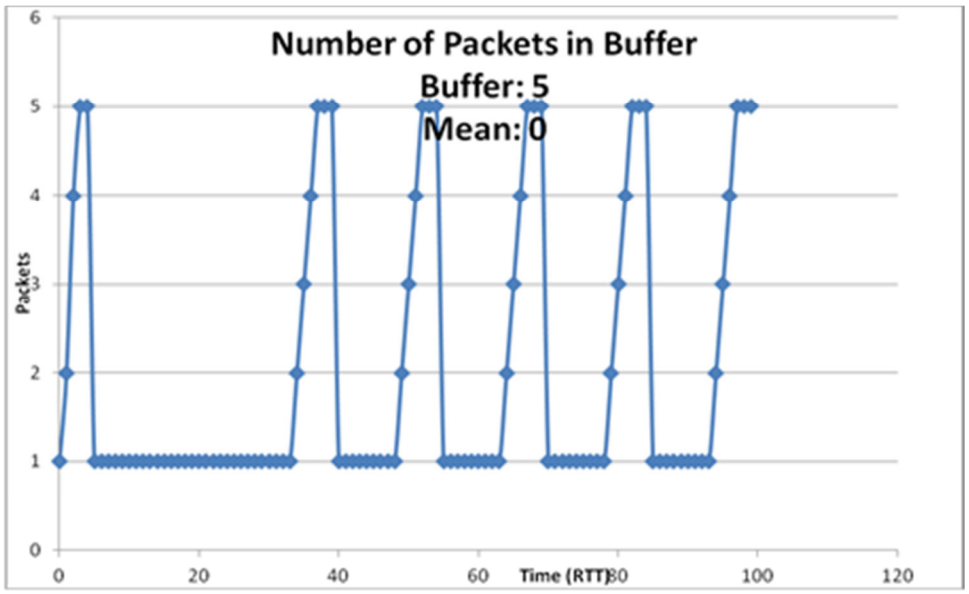
In the original code TCPSegments were moved around in the packets[] array using pass by reference. This array was passed from sender to router to receiver and each would modify the array as needed. This worked because packets were either dropped or passed through to the next class, they were not stored. Since we need to delay the TCPSegments, we need to store them in a buffer and we do not want any other class to accidently affect our copy. To do this in Java, we need to explicitly call the new keyword but to simplify matters we implemented a clone() function in TCPSegment which returns a copy of itself. In addition we added a new constructor to TCPSegment for use by the clone() function. The original constructor printed out information every time a TCPSegment was created. Because we did not want that information duplicated, we made a new constructor. In addition, we added a delay field to the TCPSegment class. We had two options for keeping track of delay, either in TCPSegment as a field or separately in a second Vector. It was cleaner from a programming perspective to make the delay part of TCPSegment so we implemented the first option.

In addition to modifying the Router and TCPSegment class' we needed to slightly alter the runner class, TCPSimulator. In this class we modified the main() function to read in mean and standard deviation as command line parameters. Overall the program takes four user specified parameters, iterations, buffersize, mean, and standard deviation. Currently the implementation does not make use of the standard deviation parameter because the exponential distribution depends only on the mean. It was included however to facilitate future work with alternate distribution functions which may depend on standard deviation.

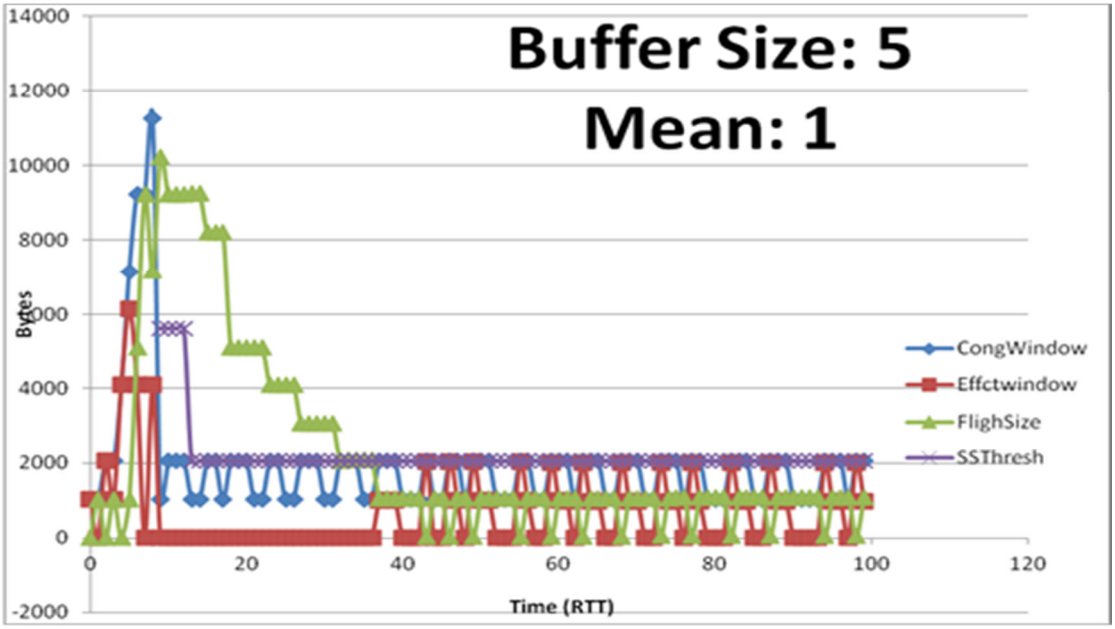
Results and Discussion

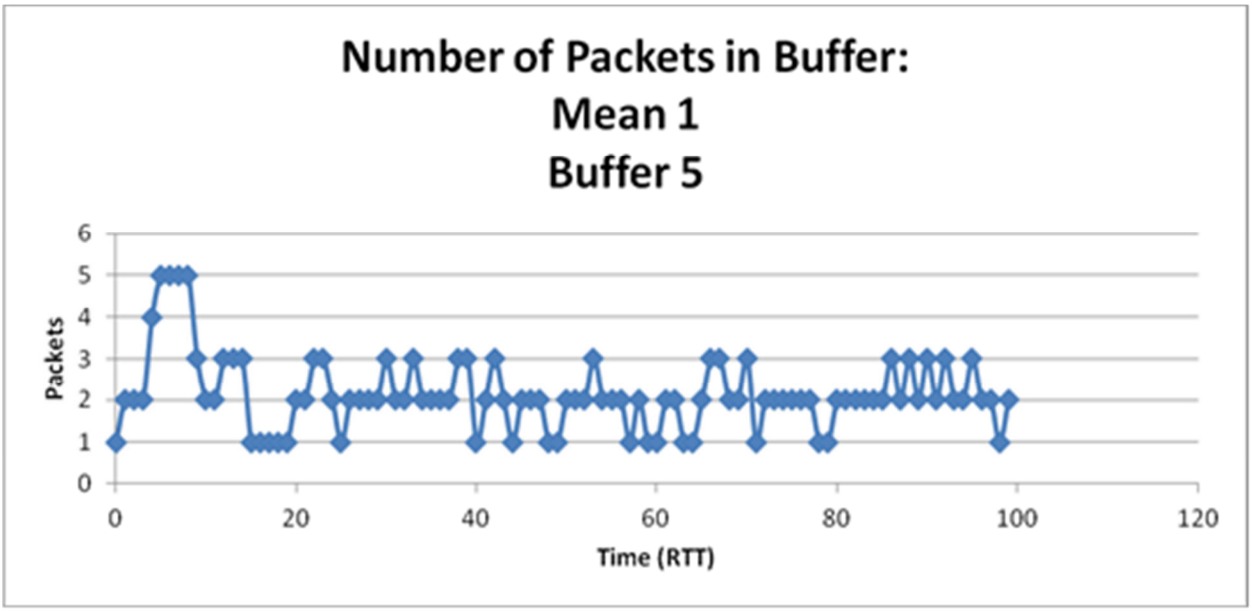
To determine the effect of adding a random exponential delay as a comparison, the mean delay is initial set to 0, with a maximum buffer of 5. It can be seen that initially the sender starts in slow start, and continues until the buffer overflows, receiving 3 dupACKs. This leads to the sender entering multiplicative decrease. Eventually after sending all the packets that were lost and the congestion window exceeds SSThresh, the sender goes into congestion avoidance. After it exceeds the buffer threshold and the sender receives 3 dupACKs, the sender again enters multiplicative decrease. This pattern continues throughout the rest of the testing round. The overflow that continues to occur throughout this pattern causes a total of 23 packets to be dropped because of overflow.



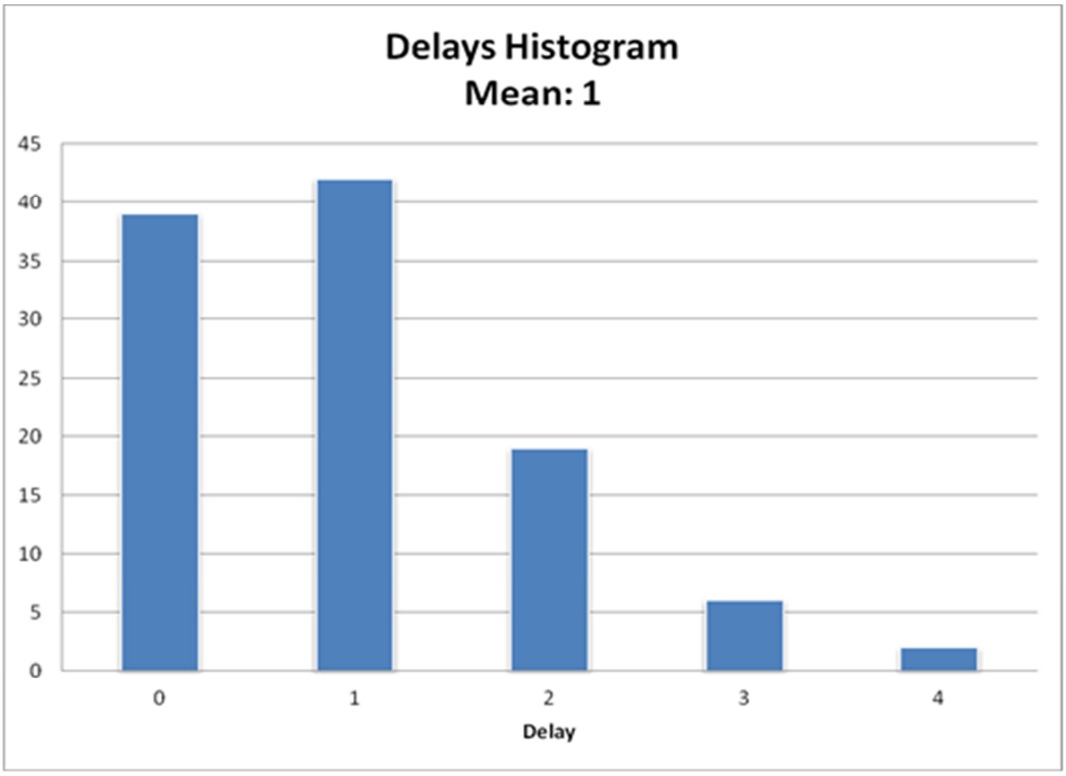


The mean value of the exponential distrubtion was then set to 1, to determine the resulting effect the packet reordering has on the utilization of the receiver. The random delay causes packets to take time to get through the buffer, and for the sender to send additional ACks for those packets, even if they are not lost. In addition, because of the random delays, the sender also timeouts because it takes such a long time for the packets to get through the router. After reaching its max buffer size of 5, the number of packets varies between 1 and 3 packets.

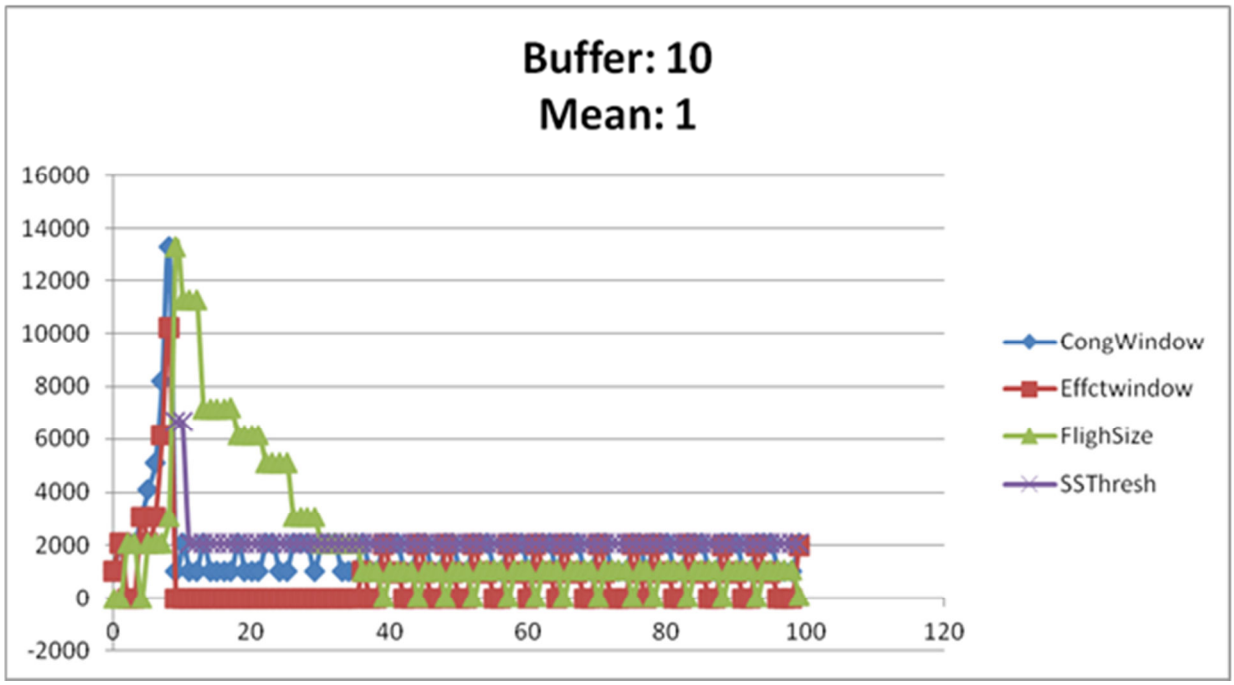


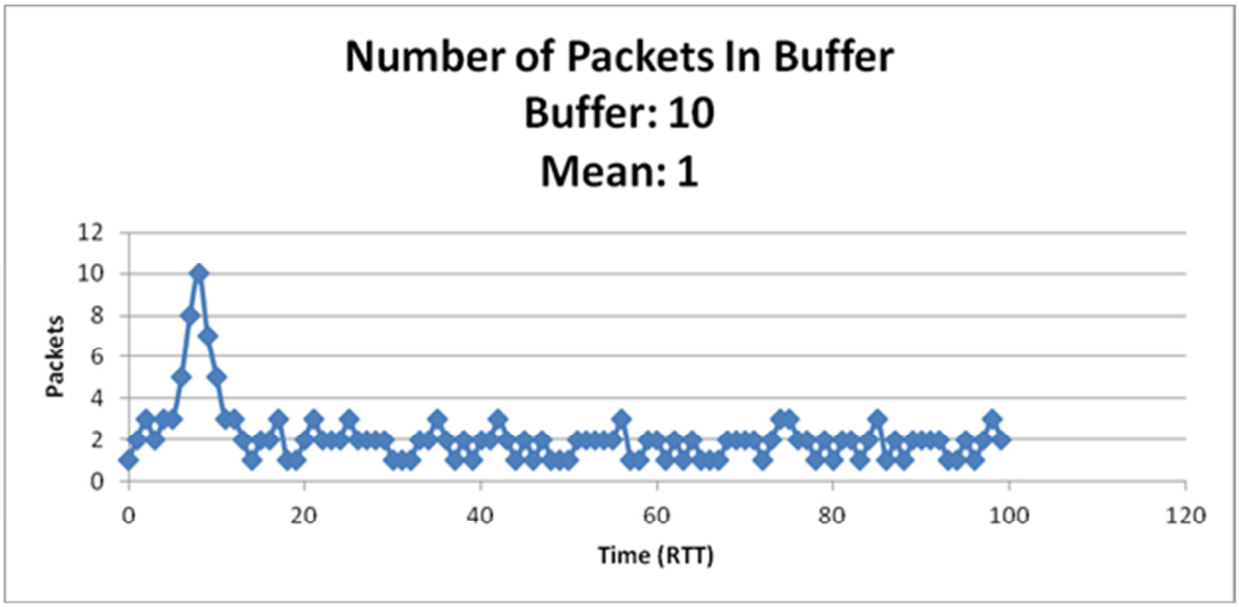


In addition, it can be seen by looking at the Delay Histogram, the distribution that occurs for this sample run. As it is supposed to, the delay histogram follows an exponential distribution. The delays of 3 and 4 that occur, greatly slow down the router, and decrease the efficiency of the system. Adding a mean delay of one causes the average utilization to drop to 4.9%, from the 25% that occurred when there is no delay. Even though only 5 packets are dropped because of router overflow, the efficiency of the system decreases. The mean delay of 2 is not tested because this will cause an even more substantial delay, and almost no packets would get through the system.

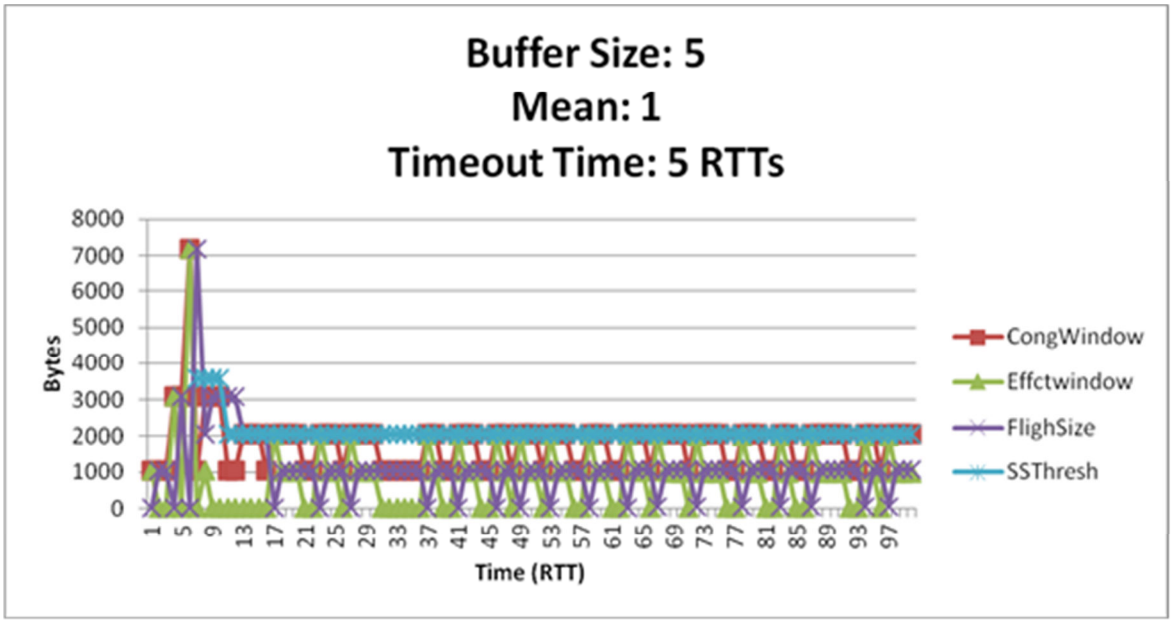


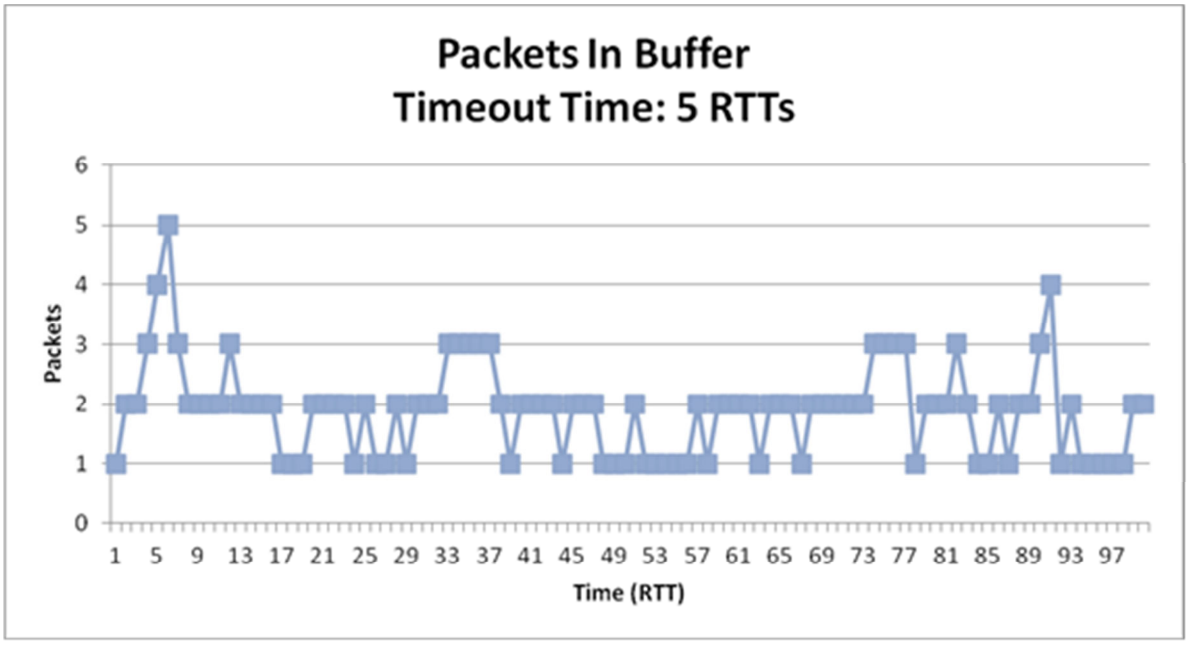
To determine the effect the buffer size has on the output, everything else is kept constant, while the buffer size is increased to 10. It can be seen that the congestion window and flight size peak at above 13000 bytes, which is higher than the peaks with a buffer size of 5. Even though there are delays in the beginning, the sender is able to begin in slow start. It soon fills the buffer, and enters multiplicative decrease. After this initial router overflow of 2, the system is not able to recover during the time tested due to the random delays incorporated. The number of packets in the buffer reaches 10 during the initial slow start but after that, it cannot get through the timeouts and the 3 dupACKS to get more than 3 packets in the system. Increasing the buffer size actually causes a drop in the average utilization of the system to 2.85%. Changing the buffer size still does not allow the number of packets in the buffer to increase. Other possible causes of this problem are the timeout length and the long delays. These cases are tested next to determine the effects on the system.



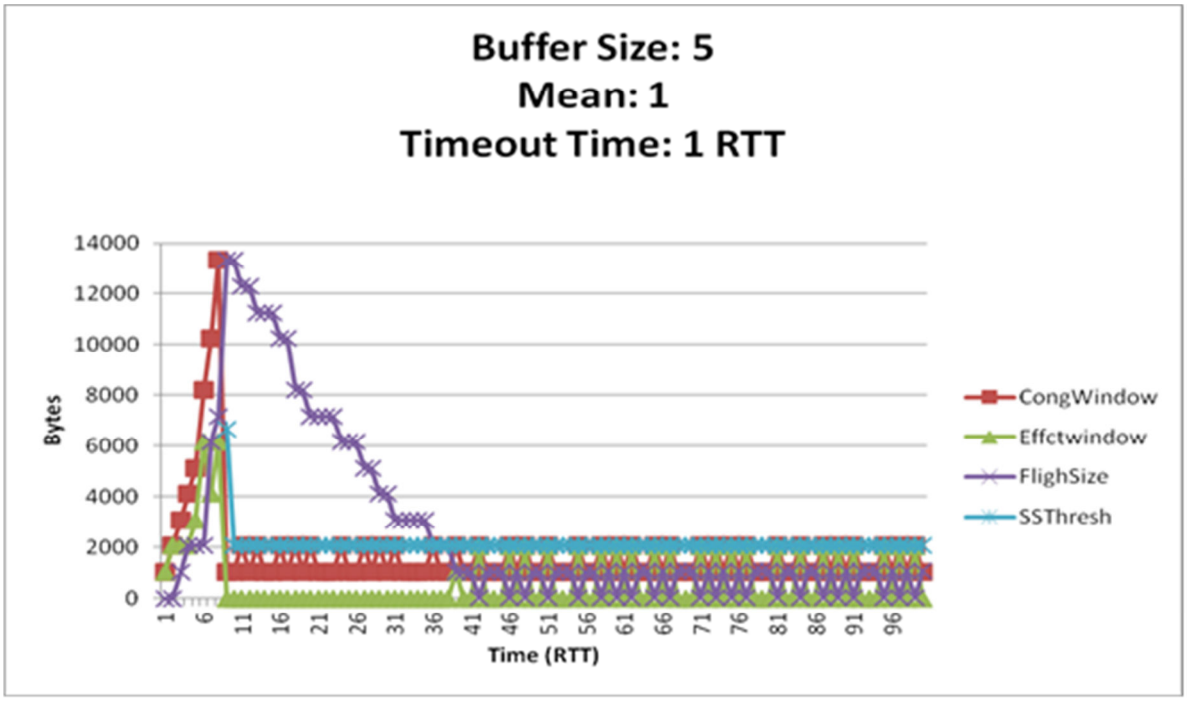


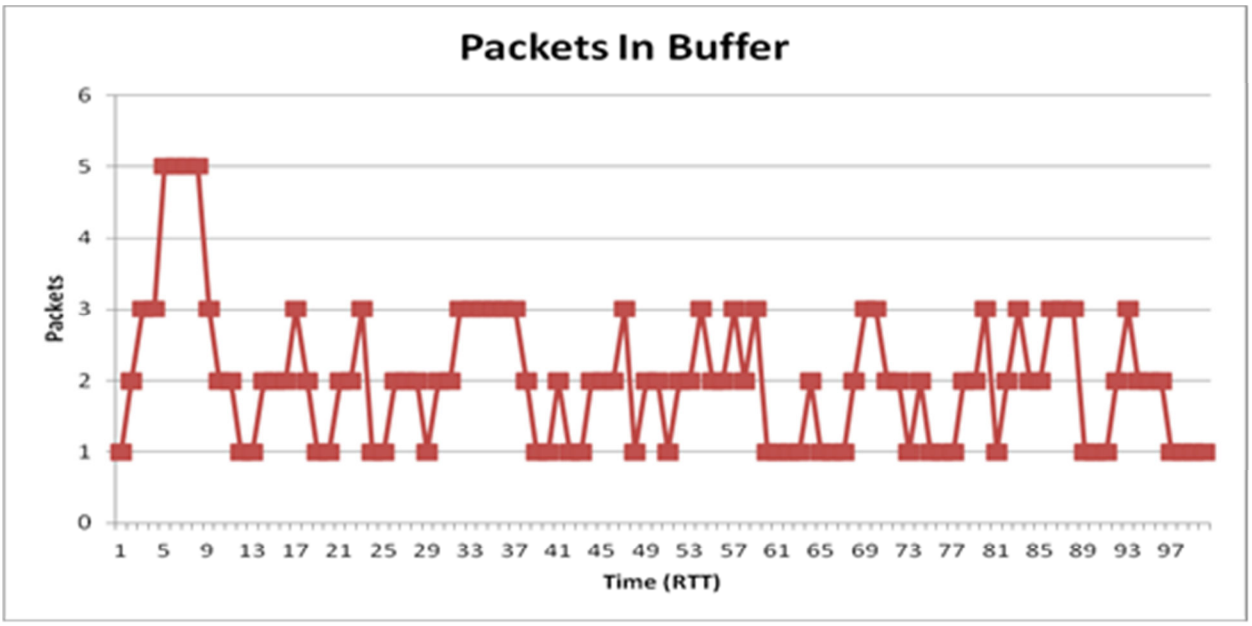
The next case tested was to increase the timeout length from 3 RTTs to 5 RTTs. The hypothesis for this variation was that the sender would not time out as easily thereby improving the utilization. It is determined that average utilization drops to 4% and after the buffer peaks at 5 packets, the number of packets in the buffer decreases and varies between 1 and 3 packets in the system for the rest of the testing period. Even though there were no time outs, there were more cases of at least 3 dupACKs. As before the random delay still disrupts the system, and does not increase the utilization.



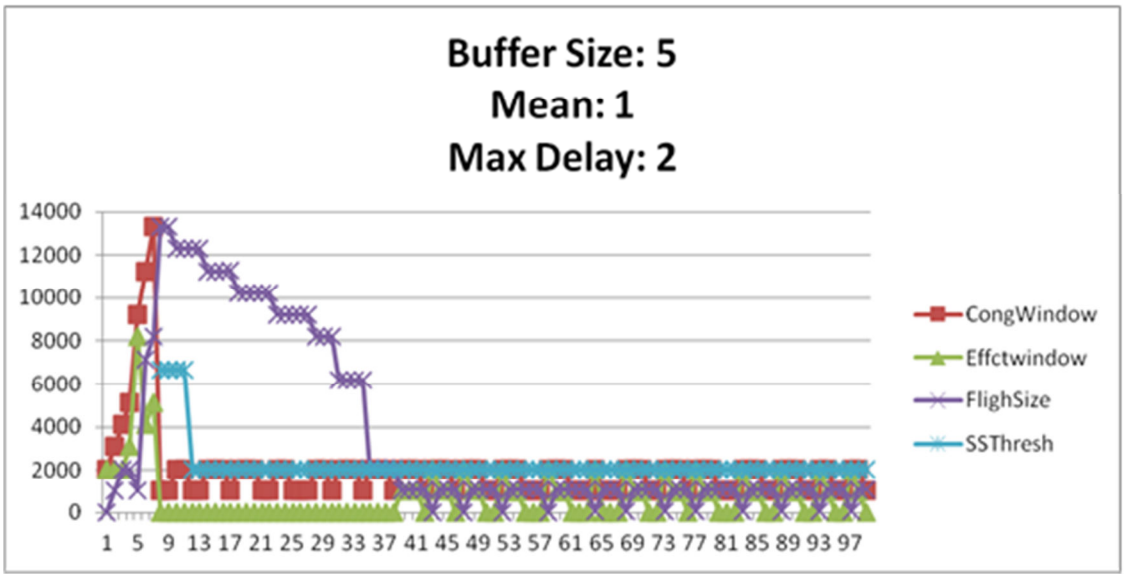


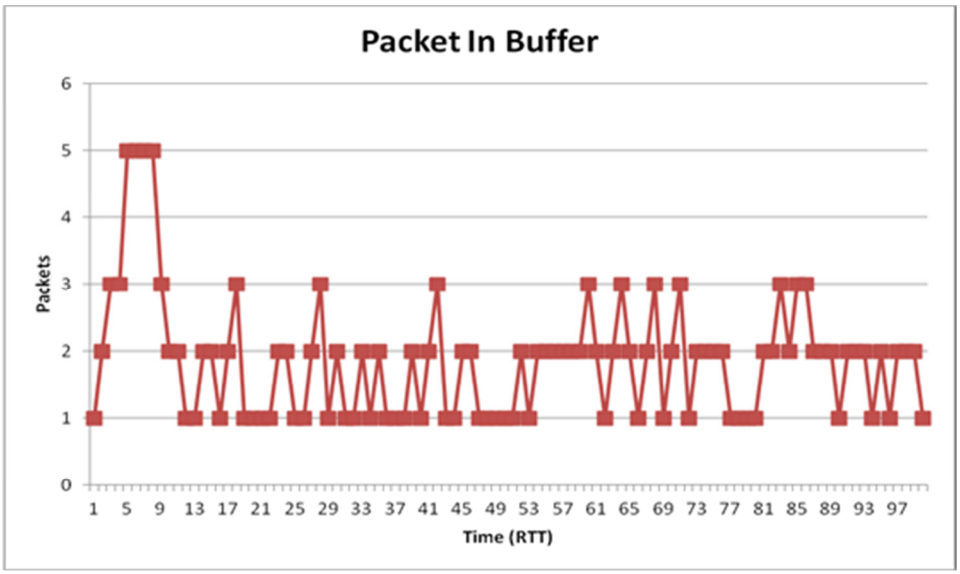
Another test case scenario is when the timeout time is decreased to 1 RTT. This caused numerous timeouts to occur in the output. Even though there were a great deal of timeouts, the sender utilization increased to an average of 7%. Instead of waiting for the packets that had a long delay and getting 3 dupACKS, a timeout occurred and the sender resent the packet. Although there is a slight improvement in utilization, the flight size plot does not vary much and the number of packets in the buffer continues to vary between 1 and 3 packets. By lowering the timeout time, the sender quickly resends the packets if the delay is too long which slightly increases the utilization.

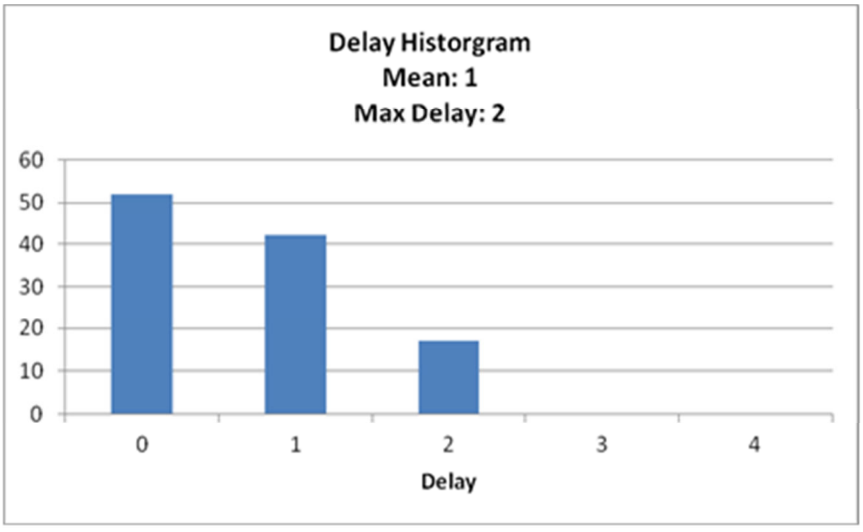




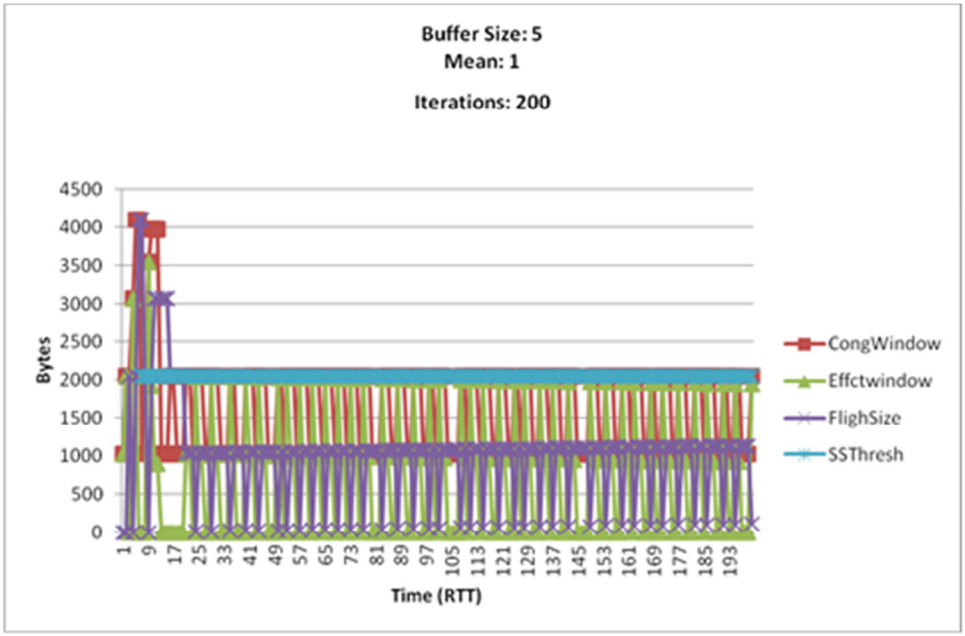
The exponential distribution caused some packet delay times to be more than the timeout length, which resulted in the packets taking too long to get through the router. For the next scenario, a cap was added to the possible delays times. If a delay of 3 or more occurred, a new delay was recalculated until a delay of less than 3 occurred. The average sender utilization only increased slightly from the original case to 5.3%. It is determined that removing the longer delays does not have a great effect on the system, and that even a short delay of 1 or 2 RTT is enough to cause inefficiency. Other than a slight increase in the utilization, there are no other noticeable differences in the response of the system.

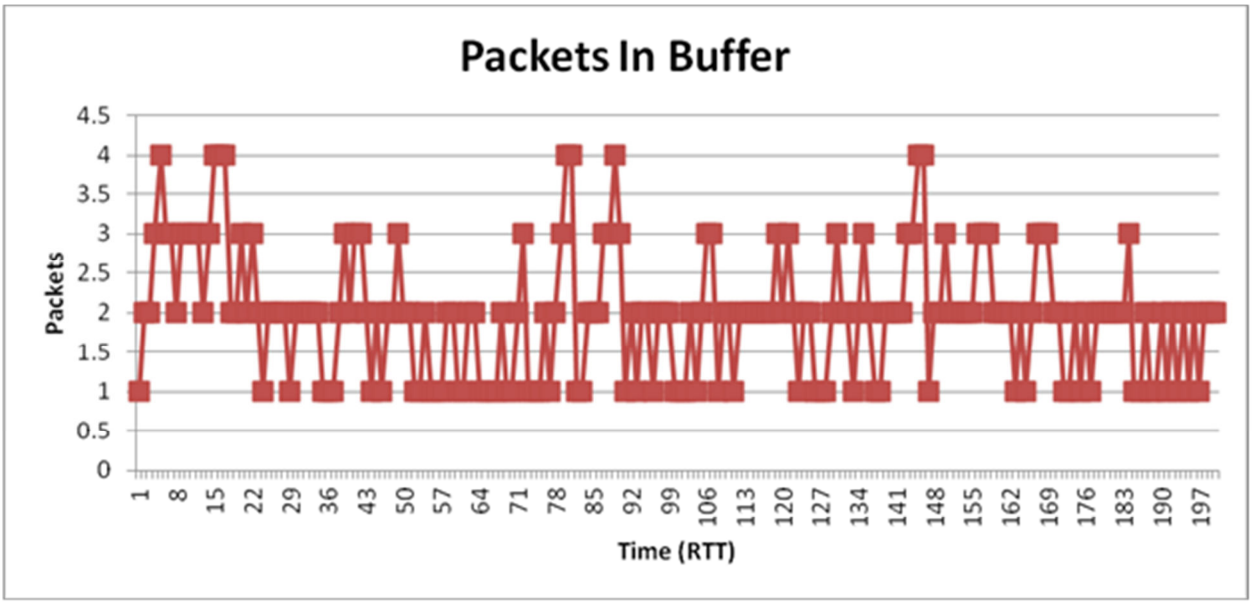

Buffer Size: 5
Mean: 1
Max Delay: 2
14000
12000
10000
8000
6000
4000
2000
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
CongWindow
Effctwindow
FlighSize
SSThresh


Packet In Buffer
Packets
6
5
4
3
2
1
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
Time (RTT)


Delay Historgram
Mean: 1
Max Delay: 2
60
50
40
30
20
10
0
0 1 2 3 4
Delay

The last design consideration that was varied, was to increase the number of iterations from 100 to 200. This did not really have a noticeable effect on most of the outputs tested however the sender utilization dropped slightly to 4.5%. What was noticeable about this sample output was that even though the buffer size is 5, the number of packets in the buffer never reaches this level. This is due to the random nature of the packet delays. In addition, the random delays allowed the number of packets to increase to 4 at other times in the period tested.





This project was able to reveal the effects of random delays on TCP Tahoe. The random delays caused the packets to be re-ordered in the buffer, which resulted in the sender receiving numerous 3 dupACKS and timeouts. The different test cases that altered different parameters did not lead to great improvements from the initial delay case. The effect of the delays was too prevalent to allow the system to approach the utilization of the case with no delay. When there was a delay, the cases all followed the same pattern of starting in slow start leading to a buffer overflow. This leads to multiplicative decrease, but because of the lost packets and random delays the sender rarely will enter additive increase. The system always has at least one packet in the buffer because the sender sends one bit keep-alive packets. In conclusion, the more realistic TCP Tahoe simulation reveals the problems that delays cause in TCP Tahoe.

References

Books

Peterson, Larry L., and Bruce S. Davie. Computer Networks: a Systems Approach. San Francisco, CA: Morgan Kaufmann, 2011.

Ivan Marsic, Computer Networks: Performance and Quality of Service, Rutgers University, Department of Electrical and Computer Engineering, 2011.

Websites

"Exponential Distribution." Lectures on Probability and Statistics. Web. 06 Dec. 2011. <http://www.statlect.com/ucdexp1.htm>.

"Exponential Distribution." Wikipedia, the Free Encyclopedia. Web. 06 Dec. 2011. <http://en.wikipedia.org/wiki/Exponential_distribution>.

Recovery, Using Fast. "Yee's Homepage | TCP/IP | Background." UCL HEP Group. Web. 06 Dec. 2011. <http://www.hep.ucl.ac.uk/~ytl/tcpip/background/tahoe-reno.html>.

"TCP Congestion Avoidance Algorithm." Wikipedia, the Free Encyclopedia. Web. 06 Dec. 2011. <http://en.wikipedia.org/wiki/TCP_congestion_avoidance_algorithm>.

Appendix

The code was run using eclipse. To import the project and change the user input values you follow the following steps:

1) Open the package explorer by going to Window -> Show View -> package explorer

2) Import the project into eclipse by going to file -> import select existing project into workspace in the general folder. Give it the root directory of the project.

3) The project should appear in the package explorer, right click on it and go to run as -> run configurations. Then, go to the arguments tab and type in the desired arguments separated by spaces, for example '100 10 1 2'. In this example 100 is the iterations, 10 is the buffer size, 1 is the mean, and 2 is the standard deviation.

Note that the value for timeout is not a user input, meaning changing it involves modifying the value of TIMER_DEFAULT in TCPSender.

Value ranges: all input values must be integers in the following range

iterations= 0-200

buffer > 1

mean >= 0

standard deviation >= 0

After running the code, the program will output two files to the project root directory, they are delay.txt and droppedpackets.txt. Delay has delay values for every packet for each iteration as well as the total number of packets in the buffer (the column labeled packets is the packet the delay corresponds to). The droppedpackets file contains how many packets are dropped during each iteration.